@MASM,L MDM.BITMOVE,BITMOVE
MASM 4R1A (840907 0128:18) 1989 Ma

UNISYS

```
00    124110111123
01    040111123040
02    101040124105
03    123124040040
      000000000220
04
```

```
00    10  16  00  00  0  000004
* *           BITS  17-0 + LC  0
01    10  00  01  00  0  000011
* *           BITS  15-0 + LC  0
02    46  16  15  00  0  004400
03    37  10  00  00  0  000014
04    01  00  00  00  0  000012
* *           BITS  15-0 + LC  1
05    10  16  00  00  0  000010
            BITS  17-0 + LC  1
                  00  0  000000
* *           BITS  15-0 + XR  S
      04  00  00  0  000013
* *           BITS  15-0 + LC  1
```

Unisys 1100/2200 MASM: Introduction, Techniques and Style Guide

Michael D. Maddox

Published 1990 by
Publications & Communications, Inc.
12416 Hymeadow Drive
Austin, Texas 78750-1896

Copyright PCI, Inc. 1990

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording or otherwise, without prior permission, in writing, from the publisher.
For permission, contact PCI, Inc.

PCI editor and production supervisor: **Michele Drolet**
Cover design: **Les Case**

ISBN **0 – 9627241 – 0 – 6**

Unisys 1100/2200 MASM:

Introduction, Techniques and Style Guide

Michael D. Maddox

This book is dedicated to my wife, Donna, and my son, Greg.

Thanks

Many thanks to the Fort Collins, Colorado, US Department of Agriculture and Unisys people who were kind enough to review the first draft: Sheryl Barr, Jan Durnin, Cindi Eichin, Mary Heimbichner, Larry Hulse, Jim Landis, Jennifer Schukar, Dorothy Shinn-Bailey, Dave Till and Rick Yancik.

A special thanks to Ted Brenneman and Dan Crosson of Chi Corporation for their especially thorough reviews of the first draft.

The technical editor was Wilhelm Tomaschik of Unisys. He is responsible for a number of corrections and clarifications. A tip of the hat to Wilhelm.

Preface

There are in the neighborhood of two thousand Unisys (formerly Sperry) Series 1100 and 2200 systems in use all over the world. Despite this large installed base, there are few if any non-Unisys materials available to assist the 1100 programmer. There are few paths open to the applications programmer aspiring to the systems specialty.

You can attend the Unisys-offered software and hardware courses; the quality is generally excellent, but the costs can be prohibitive to the small 1100 shop.

Or, you can serve an apprenticeship of sorts with an experienced systems person, if one is available and can devote a share of his time to training. In fact, it may be difficult to develop yourself as a systems type without some experienced technical guidance on hand.

I am offering a third alternative with this book: I shall lead you through the intricacies of the 1100/2200 MASM environment as if you were sitting in my office. Left on your own, you might miss:

- Some sense of how things fit together.
- Why some of the hardware and software features are necessary.
- How to use the reference manuals.

While this text does not take the place of available reference materials, it is written with the following goals in mind:

- To help you learn CPU features.
- To help you learn MASM processor features and directives.
- To help you identify key information sources.
- To give you code templates for using many instructions and ERs.
- To offer you a style guide for assembly language.

I shall assume that you have at least a couple of years experience programming in a high-level language, such as FORTRAN or COBOL, and are comfortable with Executive Control Language (ECL).

The style of the text is personal, based on my experience, and thus not always objective. I shall use examples wherever necessary to make a point clear. Source code will be in a different typeface, like this:

```
L           A0,PACKET       .
ER          READ$           .
```

Wherever I need to introduce a consideration of style, it will be in a box, like this:

STYLE: The tab settings for entering MASM code should be set at columns 11, 21, and 39 for the operation, operand, and comment fields, respectively.

The 1100/90 and 2200s use a different bit numbering convention from previous systems. The older convention numbered bits from 35 at the left to 0 on the right, corresponding to the powers of 2. The new 1100/90 and 2200 convention numbers bits from 0 at the left to 35 on the right. If you have been exposed to the old numbering convention, attempting a total immersion in the new convention may be hopelessly confusing. For that reason, this text uses both conventions. In each case, the new bit number is named first, and the old second in parentheses. For example, H1, the left half of a word, consists of bits 0 (35) through 17 (18).

I have made progressive changes in several code examples. To focus your attention on the differences, the lines which have been changed from the previous version are shaded, **like this.**

If the 2200 is not mentioned or specifically excluded, the term "1100" means both 1100s and 2200s.

Table of Contents

1 Information Sources 1- 1
- 1.1 Reference Manuals 1- 1
- 1.2 Unisys Worldwide Education 1- 2
- 1.3 Private Course Offerings 1- 3
- 1.4 Unisys-Related Trade Magazines 1- 3

Exercises 1- 3

2 MASM Programming in Context 2- 1
- 2.1 Introduction 2- 1
- 2.2 Assembly Language in the Business Context 2- 3
- 2.3 MASM Processor Inputs and Outputs 2- 5
- 2.4 MASM Element Subtype 2- 7

Exercises 2- 7

3 Basic Machine Architecture 3- 1
- 3.1 Machine Components 3- 1
- 3.2 The CPU as Center of MASM Programmer's Attention 3- 2
- 3.3 Number Systems Used by 1100s and 2200s 3- 3
  - 3.3.1 Octal and Binary 3- 3
  - 3.3.2 Octal Addition 3- 4
  - 3.3.3 Octal Subtraction - Ones Complement Style 3- 4
  - 3.3.4 Octal Conversion and Aids 3- 5
- 3.4 1100/2200 Operand and Instruction Formats 3- 6
  - 3.4.1 Data Formats 3- 6
    - 3.4.1.1 Fixed-Point Formats 3- 6
    - 3.4.1.2 Floating Point Formats 3- 6
    - 3.4.1.3 Characters 3- 7
  - 3.4.2 Intruction Word Format 3- 9

Exercises 3-10

4 MASM Basics
- 4.1 The Fine Print 4- 1
- 4.2 Symbols 4- 1
- 4.3 MASM Line Structure 4- 1
  - 4.3.1 The Label Field 4- 1
  - 4.3.2 The Operation Field 4- 2
  - 4.3.3 The Operand Field 4- 2
  - 4.3.4 The Comment Area, Column Conventions, and Continuation 4- 2
- 4.4 Labels and Location Counters 4- 3
- 4.5 Register and Partial Word Definitions 4- 4
- 4.6 Data Generation 4- 5
- 4.8 Other Basic Directives 4- 7
- 4.9 Character Strings 4- 9
  - 4.9.1 Character Set Control 4- 9
  - 4.9.2 EQUating Character Strings 4- 9
  - 4.9.3 Signed Character Strings 4-10
  - 4.9.4 String Functions 4-10

Exercises 4-11

5	Assembling and Running Your Program	5- 1
	5.1 Basic ECL Statements	5- 1
	5.2 Example ECL Streams	5- 3
	5.3 Listing Format	5- 5
	5.4 Assembly Error Flags	5- 6
6	Basic Integer Operations	6- 1
	6.1 Moving Data In and Out of Registers	6- 1
	6.2 Changing the Sign	6- 3
	6.3 Basic Arithmetic	6- 4
	6.3.1 Increment and Decrement Instructions	6- 6
	6.4 Instructions for Storing Constants	6- 7
	6.5 Multiplying Your Options: More Arithmetic Instructions	6- 8
	6.6 X and A Registers Overlap: The Plot Thickens	6- 9
	6.7 Double-Word Instructions	6-10
	Exercises	6-11
7	Register Addresses and Partial Words	7- 1
	7.1 Registers Have Addresses, Too	7- 1
	7.2 Partial Words: Carving Up Memory Locations	7- 1
	7.2.1 Quarter-Word and Third-Word j Values	7- 4
	7.2.2 Partial Word Examples	7- 5
	7.2.3 Partial Words Inoperative Between Registers	7- 5
	7.2.4 Immediate Operands	7- 6
	7.3 The $EQUF Directive	7- 6
	Exercises	7- 7
8	How to Alter Program Flow	8- 1
	8.1 Introduction to Jumps	8- 1
	8.2 Unconditional Jumping	8- 1
	8.3 Conditional Jump Instructions	8- 2
	8.4 Test Instructions Which Don't Need Registers	8- 5
	8.5 Test Instructions Which Do Need Registers	8- 7
	8.6 Within-Range Tests - TW, TNW	8-10
	Exercises	8-12
9	Extending Flexibility: Indexing, Indirect Addressing and Literal	9- 1
	9.1 Basic Indexing: XM	9- 1
	9.2 How To Address Fields in a Data Table	9- 2
	9.3 Index Register Incrementation: XI and h	9- 6
	9.4 Using an Index Register With Immediate Operands	9- 7
	9.5 LXM and LXI - Special Instructions for Index Registers	9- 8
	9.6 Indirect Addressing	9- 8
	9.7 Literals: Easy Constants	9-10
	9.8 Bitwise Fields: The $FORM Directive	9-11
	Exercises	9-12
10	Looping Instructions and Techniques	
	10.1 Why Loop?	10-1
	10.2 How to Code a Counted Loop	10-1
	10.2.1 Add and Test	10-1

10.2.2 Jump Greater and Decrement 10- 3
10.2.3 INC/DEC Loops 10- 3
10.4 Loops Without Counters 10- 5
Exercise 10- 5

11 Bitwise Instructions 11- 1
11.1 Shift Instructions 11- 1
11.2 Single Shifts 11- 1
11.2.1 Logical Shifts 11- 1
11.2.2 Circular Shifts 11- 3
11.3 Logical Instructions 11- 3
11.3.1 AND - Logical Multiplication 11- 3
11.3.2 OR- Logical Addition 11- 4
11.3.3 XOR - Logical Exclusive OR (Logical Difference) 11- 4
11.4 Bitwise Test Instructions 11- 6
11.5 MLU - Masked Load Upper 11- 6
11.6 Double Word Shifts 11- 7
11.7 LSC - Load Shift and Count 11- 7
11.8 Strategies For Accessing Bit-Width Fields 11- 8
Exercises 11- 9

12 Basic Executive Requests 12- 1
12.1 Introduction 12- 1
12.2 Exit ERs 12- 1
12.3 Memory and Register Snapshots 12- 2
Exercises 12- 5

13 Symbiont ERs 13- 1
13.1 Symbiont ERs 13- 1
13.1.1 Read Runstream Data in Fieldata - ER READ$ 13- 1
13.1.2 Fieldata Printing: PRINT$ 13- 4
13.1.3 ASCII Variations 13- 5
13.2 Alternate File Symbiont I/O 13- 6
13.2.1 Reading From an Alternate File 13- 6
13.2.2 Printing To an Alternate File 13- 8
13.3 ASCII Alternate Symbiont File I/O 13- 9
13.4 Symbiont ER Name Review 13- 9
13.5 Print Control Functions 13-10
13.5.1 Print Control Via ECL 13-13
Exercises 13-14

14 Subroutines 14- 1
14.1 Types of Subroutine Entries (Labels) 14- 1
14.2 EX: A "Mini-Subroutine" 14- 1
14.3 SLJ: "Sludge" 14- 1
14.4 LMJ - Load Modifier and Jump 14- 2
14.5 LIJ, LDJ, and LBJ 14- 4
14.6 Register Save and Restore Strategies 14- 5
14.7 Save and Restore Register Sets - SRS, LRS 14- 5
Exercises 14- 7

15 Repeated Instructions 15- 1
15.1 Block Transfer 15- 1
15.2 Search Instructions 15- 2
15.3 More Search Instructions 15- 4
15.3.1 Restarting a Search Loop 15- 4
15.4 Masked Search Instructions 15- 5
Exercises 15- 6

16 MASM Procedures 16- 1
16.1 Internal Procedures - $PROC 16- 1
16.2 The $NAME Directive 16- 3
16.2.1 Parameter References 16- 3
16.3 External and Waiting Labels 16- 4
16.4 External Procedures 16- 5
16.5 Proc File Search Order 16- 6
16.6 The DEF Directive 16- 7
16.7 Changing Location Counters 16- 8
16.8 How To Write a Proc 16-10
16.9 Listing Control 16-10
16.10 The $GO Directive 16-11
16.11 Procs and Subroutines 16-11
16.12 Some Handy Procs 16-12
Exercises 16-13

17 MASM Operators and Expressions 17- 1
17.1 Introduction 17- 1
17.2 Arithmetic Operators 17- 1
17.3 Logical Operators 17- 2
17.4 String Operators 17- 4
17.5 Relational Operators 17- 4
17.6 Single and Double Precision Control 17- 5
Exercises 17- 6

18 More MASM Directives 18- 1
18.1 $GFORM - Generalized $FORM Directive 18- 1
18.2 $DO - Repeat a Single Line 18- 1
18.3 $REPEAT - Repetitively Assemble a Range of Lines 18- 2
18.4 $ON and $OFF - Conditional Assembly 18- 3
18.5 $IF, $ELSE, and $ENDF Directives 18- 3
18.6 $ELSF and $ANDF Directives 18- 4
Exercises 18- 5

19 MASM Style 19- 1
19.1 Why Develop a Style? 19- 1
19.2 Build ECL for Assembly and Mapping 19- 1
19.3 Parameterize Everything You Can 19- 2
19.4 Modularize Functions 19- 3
19.5 Use Registers Consistently 19- 4
19.6 Limit Range of Unconditional Jumps 19- 4
19.7 Use Multiple Location Counters 19- 5
19.8 Make Labels Into Comment Lines 19- 5

19. 9 Assign Labels Systematically 19- 5
19.10 Unnamed Jump Targets and Coding 19- 6
19.11 Use Clear In-Line Comments 19- 6
19.12 Use Procs to Generate Repetitive Code 19- 7
19.13 Pipelining and Performance 19- 7
19.14 What Goes First? 19- 9

20 Basic Data Structures 20- 1
20.1 Begin With a Little Theory 20- 1
20.2 Definitions 20- 1
20.2.1 Linear and Linked Lists 20- 2
20.3 MASM Code for Data Structures 20- 3
20.3.1 Avail List 20- 3
20.3.2 Node Layout 20- 5
20.3.3 Traversing a Linked List 20- 5
20.3.4 Linked List Insertion 20- 6
20.3.5 Linked List Deletion 20- 6
20.3.6 Linked List Sorting 20- 7
20.4 Variations in Linked Lists 20-10
20.5 Summary 20-11
Exercises 20-11

21 Using Character Data 21- 1
21.1 Basics of Character Manipulation 21- 1
21.2 Character Vectors 21- 2
21.3 Changing Character Sets 21- 5
21.4 Hardware-Dependent Solutions 21- 5
21.5 Converting Fieldata or ASCII to Binary 21- 7
Exercises 21- 9

22 More ERs 22- 1
22.1 Initial Execution Status 22- 1
22.2 ER OPT$ 22- 2
22.3 ER DATE$ 22- 2
22.4 ER TDATE$ 22- 3
22.5 ER TIME$ 22- 4
22.6 Dynamic Facilities Requests 22- 4
22.6.1 Checking CSF$/ACSF$ Return Status 22- 5
Exercises 22- 6

23 Collecting Your Program or Before the Program Executes 23- 1
23.1 Check Your Assembly 23- 1
23.2 Get to Know the Collector 23- 1
23.2.1 Overview of the Collection Process 23- 1
23.2.2 Basic Collector Directives 23- 2
23.2.2.1 INcluding Files and Elements 23- 2
23.2.2.2 FRSTINing an Element 23- 3
23.2.2.3 NOTing Out a File or Element 23- 3
23.2.2.4 LIBrary Searches 23- 3

24 Executing Your Program 24- 1
24.1 How to Execute Your Program 24- 1

25 Program Error Analysis or Picking Up the Pieces 25- 1
25.1 Types of Errors 25- 1
25.1.1 Program Contingencies 25- 1
25.2 Confronting the Error Termination 25- 3
25.2.1 Problem Solving 25- 3
25.3 The Post Mortem Dump (PMD) 25- 4
25.3.1 Interpreting Instruction and Data Words 25- 5
25.4 Other Debugging Aids 25- 6
25.5 FLIT 25- 7
25.6 Recapping the Strategy 25- 7
Exercises 25- 8

Chapter 1: Information Sources

1.1 Reference Manuals

The systems programmer has a number of places to go for information to aid in his work.

The first and most obvious are the reference manuals available from Unisys. The UP manuals are the backbone of 1100 documentation. However, they are intended for reference only. In practice, the term "reference" means that you are pretty sure which instruction you want to use, but you need to check details of its operation.

(By the way, the UP numbers quoted may include revision numbers or dot numbers, like UP-9999.1. If you order one or more of these, the latest revision or dot number will not be assumed by the documentation order center. Another important note: some of the manual references in this text may change with updates to the manuals.)

Little effort is expended in reference manuals to explain the context of an instruction or ER, or why you would use it in a particular situation. The fledgling MASM coder will probably want to start with an excellent manual called the Assembly Instruction Mnemonics Supplementary Reference, or AIM for short, UP-9047.1. ("Supplementary"? It supplements the processor and storage reference manuals.) This work comes closer than any other Unisys publication to explaining in plain English how each 1100 instruction works.

As you progress in your knowledge of the 1100, you will need to check on fine points of instruction execution. Check the Processor and Storage Reference, UP-9667, Rev. 1 for the 1100/90, UP-8604, Rev. 1 for the 1100/80 and UP-9652, Rev. 1 for the 1100/70 or 1100/60. (The 2200 counterparts are listed later.)

No 1100 MASM programmer will want to be without the Exec System Software Quick-Reference Guide, UP-7824.3. This 5.5x8.5-inch-format handbook capsulizes ECL, the Executive Requests, instructions, character sets and error codes all in one easy-to-use volume.

One of the manuals you will use which is most difficult to understand is the MASM Reference, UP-8453.4. But since you can only speak to the 1100 hardware by way of MASM, there are certain parts of the knowledge contained in this manual you cannot avoid. (Other parts you *can* avoid, thank goodness.) The present book will give you a basis for understanding more advanced concepts in the MASM reference.

The granddaddy of all 1100 manuals is the Executive System Programmer Reference, UP-4144.32 (for Level 41R3, it is UP-4144.38; UP-11563.5 covers ECL). These weighty tomes are the last word on 1100 ECL, Executive Requests and other aspects of dealing with the operating system.

You should have your own copy of most of the manuals listed above. The Processor and Storage Manual is the main exception to this rule; you'll only need the one for the system you plan to support. And, too, with a few exceptions, the 1100 instruction set is pretty much identical between the systems; for your purposes as a beginner, almost any of the three will do.

Other manuals you may need are:

- UP-11286 Rev. 1, the 2200/200 System Processor Complex, Quick-Reference Guide.

- UP-11275, the 2200/200 System Processor Complex, Programmer Reference.
- UP-12451, the 2200/400 System Central Complex Programming Quick-Reference Guide.
- UP-11452, the 2200/400 System Central Complex, Programmer Reference.
- UP-8731, the Data Structures Programming Reference Manual (Exec Level 38).
- UP-8435.3, the FLIT Processor Operations and Programming Reference Manual Volume 1: Program Mode (Level 7R1).
- UP-13424, the FLIT Processor Operations and Programming Reference Manual Volume 2: System Mode (Level 7R1).
- UP-8486.16, the Exec System Software Administration and Support Reference Manual (Level 41R3). (For Level 39R3, this was known as the Operating System Installation Reference.)

Here is a recap of the manuals mentioned earlier in the text, with their full formal titles:

- UP-9047.1, the Assembly Instruction Mnemonics Supplementary Reference.
- UP-9667 Rev. 1, the 1100/90 Systems Processor and Storage Reference.
- UP-8604 Rev. 1, the 1100/80 Systems 4x4 Capability Processor and Storage Reference Manual. (The term "4x4 Capability" simply refers to the 1100/80*A* , the predominant 1100/80 version. The original 1100/80 was limited to a 2x2 configuration.)
- UP-9652 Rev. 1, the 1100/70 Systems Processor and Storage, Hardware, Programming Reference Manual.
- UP-7824.3, the Exec System Software Quick-Reference Guide (formerly the Hardware/ Software Mini-Reference Executive Level 39R3).
- UP-8453.4, the Meta-Assembler (MASM) Programming Reference Manual.
- UP-4144.32, the Executive System (Exec) Level 39R3, Programmer Reference.
- UP-4144.38, the Exec System Software Executive Requests Programming Reference Manual (Level 41R3).
- UP-11563.5, the Exec System Software Executive Control Language Operations and Programming Reference Manual (Exec Level 41R3).

1.2 Unisys Worldwide Education

At this writing, the 1100 Systems Programming curriculum was being revised. Please get in touch with your Unisys representative for up-to-date information.

1.3 Private Course Offerings

The author is not familiar with 1100 systems education offerings of third parties. If this is the route you decide to take, you may find ads for them in the 1100-oriented trade magazines, or you can check with your employer's training coordinator.

1.4 Unisys-Related Trade Magazines

There are excellent systems-oriented articles in the magazine *Unisys World*. You may want to check out *Unisphere Magazine*, as well.

Exercises

E1-1. Look over your collection of Unisys reference manuals. Do you have at least the MASM reference, the Processor and Storage reference for your machine and the Exec PRM or ER reference?

E1-2. Obtain a copy of the latest Unisys systems education offerings. See where you fit in relation to your career plans.

Chapter 2: MASM Programming In Context

2.1 Introduction

Before we plunge into a detailed discussion of how to code assembly language on this machine, it would be a good idea to position 1100 MASM programming in relation to other languages and machines.

Terms such as CPU, CAU and others are used in the history which follows. The Central Processing Unit of an 1100 system has also been known by these terms:

- Command and Arithmetic Unit (CAU).
- Instruction Processor (IP).

The functionality from a MASM programmer's standpoint has not changed. Similarly, the input/output processing section of 1100 systems has been known variously as:

- Input/Output Access Unit (IOAU).
- Input/Output Unit (IOU).
- Input/Output Processor (IOP).

There have been variations in the functionality of these components, but unless you get involved in Exec (OS 1100) internal maintenance, you don't need to worry about them.

The first 1100 system, the 1101, was built especially for an agency of the U. S. Government and its "Project 13". (Thirteen in binary is 1101_2.) It had little in common with today's 1100s.

The 1100 as it exists today first appeared on the market around 1965 as the 1108. The machine was originally built with scientific users in mind, but it was also sold to commercial users. For a relatively brief time early in its history, the 1100's operating system was Exec II, capable only of batch work, one job at a time.

The next operating system, Exec 8, has survived in spirit, if not in detail, to the present day as OS 1100. Nearly every complex of the operating system has been rewritten since then. And only one of the major components which could be connected to an 1108 is presently supported (the Communications Terminal Modular Control, or CTMC, reborn as the General Communications Subsystem, or GCS).

The 1108 itself was a 36-bit word machine, as are present-day 1100s. It was limited to 262,143 words of core memory. (As you can imagine, the resident operating system did not occupy 200K of that memory!) Each 65K-word memory module stood almost as high as a refrigerator and half-again as wide. The basic memory cycle time was 750 nanoseconds; this was the last machine for which instruction timing was published. As we discuss features of present 1100s, we shall go into more detail of 1108 design.

(One feature which made the 1108 faster was the extensive use of magnetic drums. The fastest, type

0432, had an average access time of 4.3 milliseconds, which compares favorably with cached disk units of today. The capacity of 262,143 words per unit was limiting, however. The slowest, the Fastrand, at 60 milliseconds access time, was not fast, but served as an organizational concept preserved in today's software.)

The 1106, introduced a few years after the 1108, was a slowed-down, reduced-price version of the 1108, aimed mostly at the educational market. A later version could be equipped with 512K of memory, a feature which made up somewhat for the reduced speed.

The year 1973 saw the introduction of the 1110, with up to 1.5 million words of memory. The memory was divided into 512K of "primary" (fast plated-wire) memory, and 1 million words of "extended" (core) memory, equivalent to 1106 speed. The 1110 introduced hardware bank switching instructions, byte instructions and the first 4x configuration, four CAUs and four IOAUs. The 1110 was the last machine which used two different types of memory hardware for main memory.

The debut of the 1100/80 in 1977 saw a new numbering scheme for the 1100 series. The 1108 was given new memory and renamed the 1100/20. It was also given the ability to address 512K, although programs could not span the midpoint. The 1106 went through a similar resurrection as the 1100/10.

Likewise, the 1110 was given new memory and a new life as the 1100/40. A six-processor special version of this machine was sold to NASA for Space Shuttle mission simulation.

Overshadowing the 1100/40 announcement, though, was the 1100/80, later dubbed the 1100/80A. Like the 1110 and 1100/40, the 1100/80A could be configured with up to four CPUs and IOUs, and offered byte instructions to speed Cobol executions. The /80 was Sperry's first machine equipped with cache memory and up to 4 million words of main memory. But it could also be equipped with a Scientific Accelerator Module (SAM) for faster floating point performance. And it could be mated with an array processor for heavy-duty number crunching, such as seismic processing used in oil exploration.

While the 1100/80 and prior machines used hard-wired instructions, the first microcoded machine from Sperry was the 1100/60, introduced in 1979. The /60 made available a new level of price/performance and was Sperry's most popular machine. The optional Extended Instruction Set (EIS) used microcode to speed up character handling; Cobol and DMS-1100 got a major boost in performance. The /60 also offered three levels of cache memory: none (C models), 2K (E models) and 8K (H models).

Like the 1108, the 1100/60 was given new memory. The rechristened product, dubbed the 1100/70, increased the maximum main memory from 2 million to 8 million words. A new entry-level cacheless B model was unveiled, as well.

The top of the 1100 line, the 1100/90, was brought out in 1984. With a maximum memory of 16 million words and EIS standard, the maximum 4x configuration is rated at a 25-50x increase in performance over the old 1108. In addition, the /90's architecture introduced a new extended mode which will allow basing more and larger data banks than in basic mode.

The System 11, also known as the Mapper 10, Chaparral, or 1100/50, squeezed the central complex of memory and CPU into a box smaller than a home freezer. Physically, it was the smallest system which could run OS 1100 to date.

The first 2200, the /200, put the main CPU logic onto just a handful of integrated circuits. As Unisys' response to the challenge of "departmental computing", it can run in a normal office environment. The 2200/400 has become the new midrange offering. With the low end faster than a /70, and the maximum configuration faster than a /91, it promises to become the standard machine for many Unisys customers. The 2200/600, rated at twice as fast as the 1100/90, supports multi-system complexes for the most demanding transaction-processing requirements.

Yet wherever the 2200 series takes us technically, the features are sure to be built on the solid foundation of the 1100 series architecture.

2.2 Assembly Language In The Business Context

Please refer to Figure 2-1, The Systems Context. This diagram should help you locate yourself and your programming "on the map". The 1100 architecture is one application of digital logic design, which in turn is based on solid-state physics and the vacuum state of quantum physics. The next step up is machine language, the pattern of binary 1s and 0s recognized as instructions by the hardware.

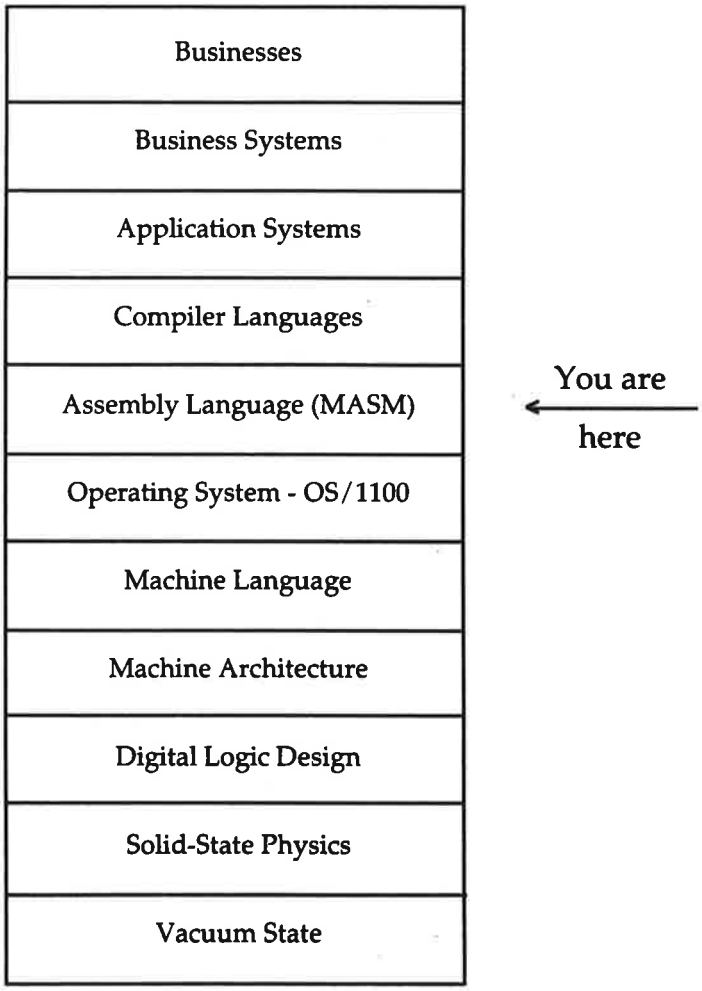

Figure 2-1. The Systems Context

Forming a useful environment for user programs to execute in is the job of the operating system, OS 1100. It loads and gives control to user programs in an orderly fashion, protecting them from each other and themselves. It defines a common, easy-to-use interface to the input/output devices attached to the system. (Imagine what a mess it would be if every program had to include its own set of machine-level input/output routines for tape drives, disk drives and so forth, let alone the problems of getting them to cooperate! And forget such niceties as remote communications.)

MASM itself, an acronym for Meta-ASseMbler, is a way of expressing machine language programs

in a human-readable form. (We refer to both the language MASM and the processor MASM with the same acronym, but the meaning should be clear from the context.) The operating system is programmed in MASM primarily, but you can write user programs in MASM, so the order of these two layers is a bit muddled.

Compiler language systems, such as ASCII Cobol and ASCII Fortran, are coded in MASM or its predecessor, ASM. In many cases, compilers may create assembly language as an intermediate step in compilation. There is another compiler language used exclusively for software development, PLUS, not shown in this hierarchy. PLUS stands for Programming Language for Univac (Unisys) Systems. PLUS is supported only for Unisys-developed products, so using it for customer-developed systems is discouraged. (OS 1100 was described as coded "primarily" in MASM; PLUS code is the major exception.)

Continuing upwards through the hierarchy: Application systems are coded in high-level (compiler) languages, which are in turn used in business systems, such as the work of an accounting department.

This hierarchy may seem trivial at first, but the point is that *no program exists in a vacuum*. The choice of a language for a given program must take into account a number of factors:

- Execution efficiency and speed.
- Development time (calendar days in the schedule).
- Development resources (CPU/memory available to compile).
- Human resources (skill sets of available programmers).
- Language features versus job requirements.
- Maintenance resources and skill sets.

You can probably add more items to this list.

In the vast majority of business systems, Cobol is the language of choice. And Fortran is unexcelled for scientific programming, "number crunching". MASM is the choice when specific hardware or operating system features must be accessed, such as processor call specifications (@PROG SPEC1). Going back to the points in the list above, here are factors to keep in mind before using MASM:

- For floating-point manipulation, stick with Fortran. Fortran can optimize object code at least as well as, often better than, an experienced assembly language programmer. If there is a great deal of character handling, stick with Cobol, for analogous reasons. A MASM subroutine can often extend the reach of Cobol and Fortran when necessary.
- For the same amount of raw logic, MASM is easily the slowest to code.
- The MASM processor is somewhat smaller than the Cobol or Fortran compilers, but the size is seldom a constraint.
- MASM programmers are harder to find or take longer to train than Cobol or Fortran programmers. (Some say they are more temperamental, too, but don't believe it.)

- Operating system calls other than I/O are more straightforward in MASM than in Cobol or Fortran, or may not be accessible any other way. Examples are two-way console communications, interrogating the run's Program Control Table, or multiple-activity operations. (These are advanced topics not covered in this book; Cobol provides indirect support for many of the ER functions described in this book. Check Section 18.8 of the ASCII Cobol reference, UP-8582.2.)

- A MASM programmer (you) may be present now to develop the program. But unless the business is committed to MASM support, the program which solves a problem now may become a problem itself later, an "orphan".

In other words, MASM is useful for solving certain kinds of programming problems, when the factors above are considered. But it will never take the place of our old favorite high-level languages.

2.3 MASM Processor Inputs and Outputs

First, let's review the basic flow of programs from source to execution in the OS 1100 system. (Please refer to Figure 2-2.) Program language statements in symbolic or "source" elements are presented to a language processor, either a compiler or an assembler. The processor creates a relocatable element, which consists of the following:

- Instructions
- Data
- External references (symbols used by, but not defined in, the relocatable)
- External definitions (symbols to be made available to other program units)

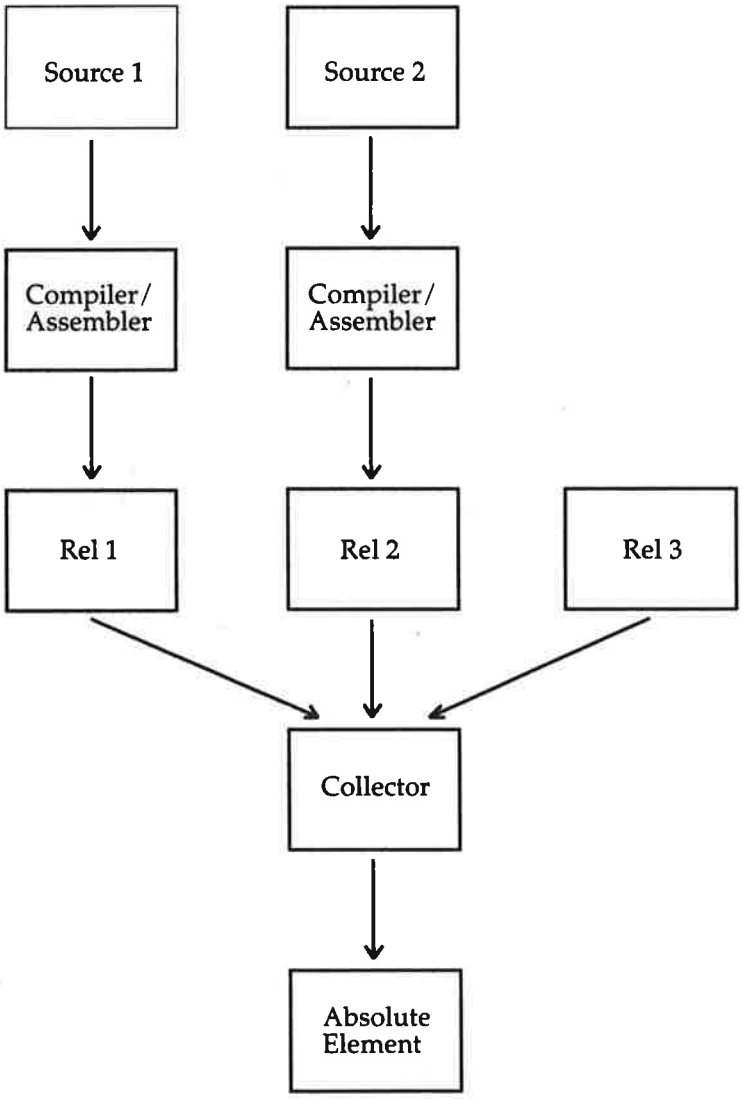

Figure 2-2. 1100 Program Element Types

- Control information (location counter lists, sizes, etc.)

The next step is performed by the collector, @MAP. (MAP stands for Memory Allocation Processor.) The collector performs these steps:

- Finds all the relocatables specifically requested by the user.
- Finds all the relocatable libraries available by default or requested by the user.
- Matches up external references and external definitions among the user-specified relocatables.
- Finds external definitions needed to satisfy user references in the relocatable libraries.
- Constructs an absolute element from the list of relocatables built in preceding steps, stacking them one after another.
- Writes the absolute element to the user-specified output file.

The absolute element itself consists of a list of memory areas, each of which will be contiguous in the final, loaded program. Each area is described by an access control word (ACW), showing its final location at execution time and its location in the absolute element.

(By the way, if you want to learn more about program file structure or the formats of relocatable or absolute elements, find the nearest copy of UP-8731, Series 1100 Data Structures.)

Next, refer to Figure 2-3 (on facing page). In its simplest form, the MASM processor accepts a source program as a symbolic element and creates a relocatable output element. Many programs require external procedures defined by the Procedure Definition Processor, PDP, from other symbolic elements. MASM turns to these locations for procedure definitions:

- MASM$PF, MASM$PFn, and ASM$PF (names attached by @USE statements).
- The use names PROC$, MASM, SYSLIB, and RLIB$.
- The source input file.
- The source output file.
- The relocatable output file.
- SYSLIB*SYSLIB.
- SYS$*RLIB$.

(The exact search order is described in the MASM manual, in section 2.4.2.) Procs can also be created by using MASM in definition mode. You can read about definition mode in the MASM manual, chapter 12.

2.4 MASM Element Subtype

An important way of documenting your work is to use the correct element symbolic subtype when you create a source element with an editor. That way, when someone (possibly you) prints the table of contents of the program file, the element will immediately distinguish itself from the others. MASM, SSG, Cobol, and Fortran have their own subtypes.

To set the symbolic subtype for MASM, use the CTS command "ASSUME COM MASM" or EDIT-1100 equivalent while editing the element. Make sure that the updated element is put back. That is, in CTS, don't forget to REPLACE (REP) the element.

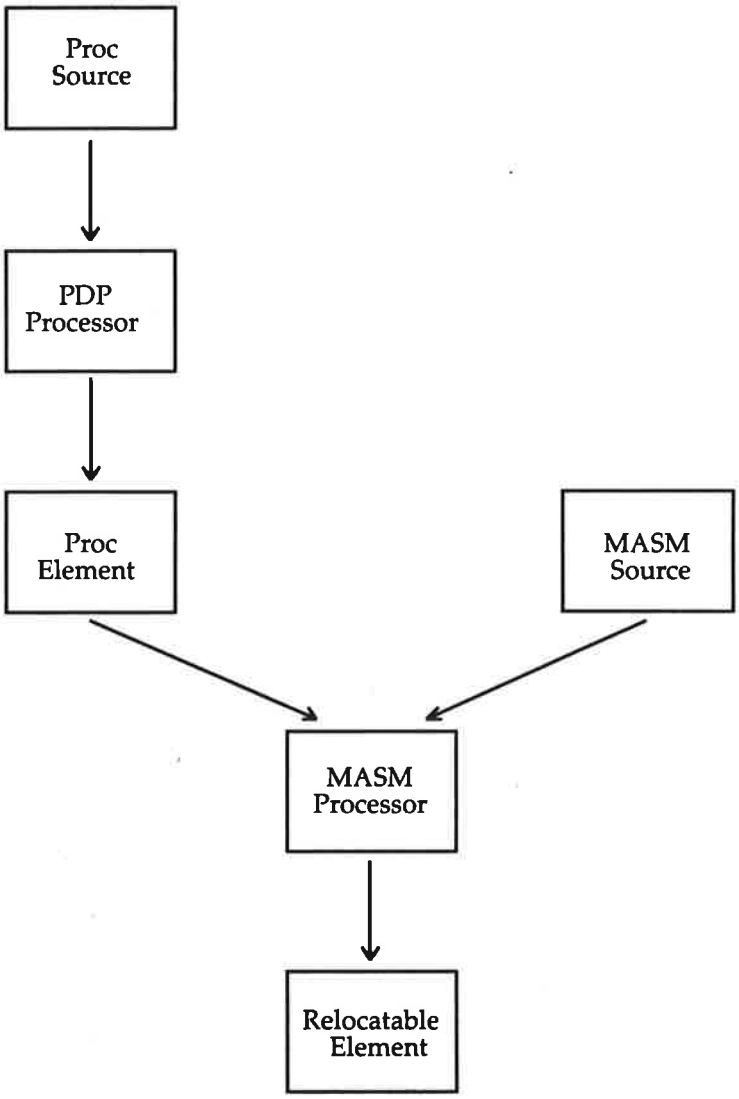

Figure 2-3. MASM in Focus

Exercises

E2-1. Discuss with your supervisor your plans to learn MASM. Ask how MASM fits into your organization's applications or support environment.

E2-2. Look up MASM's procedure search order in the MASM reference manual.

E2-3. For "extra credit", look up the element subtypes for MASM and MASM procs in the Data Structures reference.

Chapter 3: Basic Machine Architecture

3.1 Machine Components

As a MASM programmer, you should be aware of the units which your instructions command. (Refer to Figure 3-1. "Other" includes communications systems, for example.) The CPU retrieves instructions from memory, then interprets and acts on them. (Memory is also called storage, main storage, or "core", after the magnetic core memory used in second-generation computers.) The action taken by the CPU or Instruction Processor (IP) varies according to the instruction type:

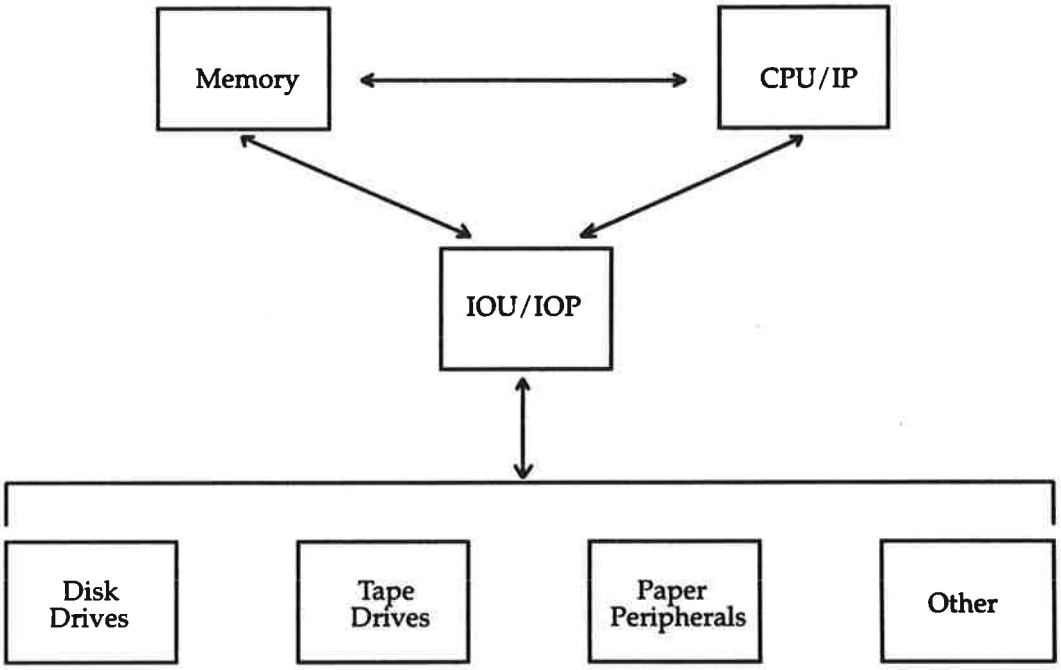

Figure 3-1. Series 1100 Components

- Some instructions involve only the CPU, such as adding two registers or setting a control flip-flop.
- Some instructions fetch data from memory and transfer it to the CPU.
- Some instructions transfer data from CPU to memory.
- Some instructions operate directly on memory locations.
- Some instructions transfer control (that is, jump).
- Other instructions indirectly cause transfer of data from memory to or from a peripheral device, that is, input-output (I/O).

The Input/Output Processor, also known as an Input/Output Unit, controls the transfer of data from memory to peripheral, once told to do so by the CPU. The details of machine-level input/output are a complex subject, beyond the scope of this book.

3.2 The CPU as Center of MASM Programmer's Attention

Since the CPU component controls the rest of the system, it pays to know more about the CPU than any other component. While the rendering of CPU features into hardware has varied between implementations (transistor-transistor logic versus emitter-coupled logic, small-scale integration versus ultra-large-scale integration, and so forth), as a programmer you can count on the basic functions not to vary. (This text will refer to all the 1100/2200 central processors, IPs or whatever, as CPUs.)

Every 1100/2200 machine has three types of registers available:

- Sixteen accumulator (A) registers, A0 through A15. With these registers, you can do fixed-point and floating-point arithmetic, test instructions, logical operations (AND, OR, XOR) and others.

- Fifteen index (X) registers (four of which overlap the A registers). You can use these registers to address tables, subroutine arguments, return points and locations over 65K. Indexing is covered in Chapter 9. You can also do addition and subtraction with them.

- Fifteen counter (R) registers, R1 through R15. These registers have the most limited function of the three types; they cannot do normal arithmetic. They serve well, though, used with Jump Greater and Decrement instructions. For certain instructions, R1 doubles as the repeat count register and for others, R2 doubles as the mask register.

More fundamentally, the CPU must keep track of the location of the most-recently interpreted instruction and locations of the current program banks. (Banks are contiguous ranges of memory within a program.) The names of the hardware registers which do these jobs have changed with almost every new 1100 system.

- The 1106, 1108, 1100/10 and 1100/20 each used a Processor State Register (PSR) and a Storage Limits Register.

- The 1110 and 1100/40 used a Main PSR and a Utility PSR, plus extensions.

- The 1100/60, 1100/70 and 1100/80 use 4 Bank Descriptor Registers.

- The 1100/90, System 11, and the 2200s have 32 Base Registers. These systems have two addressing and instruction modes, basic and extended. In basic mode, 4 of the base registers perform the functions of the 1100/80's BDRs. Extended mode is beyond the scope of this text.

Even though the hardware registers may be named differently between machines, a Jump instruction still alters the program address counter, a Load I-Bank Base and Jump (LIJ) still switches banks, and so forth. Wherever possible, this text will use generic terminology (such as CPU or program address counter), noting individual system differences only where significant.

Besides the registers accessible to the programmer, the CPU also contains hundreds of registers needed to do its job. For example, there are a number of registers used to break apart an instruction. Other registers are used to perform arithmetic operations. On a microcoded machine, an instruction operation code points to a sequence of microinstructions; there are counters to step through the microinstructions. There are also registers and counters accessible only to the Exec, such as the quantum timer, the realtime clock and the dayclock.

3.3 Number Systems Used By 1100s and 2200s

3.3.1 Octal and Binary

One inescapable characteristic of 1100s and 2200s is the use of binary arithmetic. In binary (base 2) arithmetic, suited well to on-off devices such as transistors, there are only two values, 1 and 0, equivalent to "on" and "off". Just as in decimal arithmetic, we associate positions of digits in a binary number with powers of the base. The rightmost digit is associated with the 0 power of 2, that is, 1. The next digit is associated with the first power of 2, that is, 2, and so on.

We denote a binary number by a subscript 2, for example, 01110_2 (equal to decimal 14).

To convert a binary number to its decimal equivalent, multiply each digit by the power of 2 associated with it. For example, convert 1010_2 to decimal:

```
1 x 2³ = 1 x 8 =      8
0 x 2² = 0 x 4 =      0
1 x 2¹ = 1 x 2 =      2
0 x 2⁰ = 0 x 1 =      0
                      -
                     10
```

Adding binary numbers is easy:

```
 11          (Carry)
 101110      (Addend)
+011001      (Addend)
-------
1000111
```

Add the corresponding digits from right to left. Each digit column consists of two digits (from the two operands) and a carry digit from the operand column to the right. For each column, the sum may be 0, 1, 10 or 11. If the sum is greater than 1 digit (10 or 11), the leading 1 becomes the carry for the column to the left.

Carrying around unwieldy 18- and 36-bit binary numbers could get tiresome. For convenience, we group triads of binary digits (beginning with the rightmost) as octal digits:

Binary	Octal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

With practice, you will be able to look at an octal digit and immediately tell the binary digits which comprise it. (Hint: memorizing them is a good idea.)

To save putting the subscript 8 after every octal number, we precede octal numbers with a leading zero. This is unambiguous and is the same format recognized by MASM.

The 1100 CPU arithmetic logic performs a curious type of arithmetic, called ones complement (as opposed to "twos complement"). Ones complement is defined this way:

- The leftmost bit is the sign. If clear, the quantity is positive. If the sign bit is set, the quantity is negative.
- If the quantity is negative, bits other than the sign are the reversed from their positive equivalent. For example, plus 5 is 000000000005, and negative 5 is 0777777777772.
- Plus and minus zero are separate values. Plus zero is all zero bits (000000000000); minus zero is all one bits (0777777777777).

3.3.2 Octal Addition

To add octal numbers, just align the columns as in decimal arithmetic. If the total of the digits in a column is greater than 010 (decimal 8), carry a 1 to the next column. Here is a simple example, adding 02743 and 05114:

```
carry:     11
addend:  002743
addend:  005114
         ------
sum:     010057
```

3.3.3 Octal Subtraction - Ones Complement Style

To subtract one number from another, the CPU flips the bits of the number to be subtracted (creating the ones complement) and adds the two quantities. Subtracting 02113 from 05627 involves the steps shown on the following page:

```
subtrahend:   002113
complement:   775664
minuend:      005627
difference: 1003513   (End-around carry leftmost 1 bit.)
sum:          003514
```

Of course, you don't have to do subtraction this way, using end-around carry. You can do octal subtraction much the same way as decimal subtraction:

- Borrow from the next digit left if a digit you're subtracting is larger than the corresponding digit being subtracted from.
- If the quantity being subtracted is larger than the quantity being subtracted from, just subtract the smaller from the larger and change the sign of the result.

For example, 101 decimal minus 512 is -411 (512 minus 101 is 411, which has its sign flipped). In octal, 0221 minus 0623 is negative 402, or 0777777777375.

Subtracting a number from itself illustrates an oddity of ones complement arithmetic:

```
minuend:      002113
complement:   775664
subtrahend:   002113
difference:   777777
```

Clearly, subtracting a number from itself should result in a difference of zero, yet we seem to have created a very large number. In fact, a 36-bit quantity of all one bits is defined as -0 (minus zero). It behaves as zero in arithmetic, but its sign bit is set. (A word of all 1 bits is still different from all zero bits in logical operations such as AND or OR.)

3.3.4 Octal Conversion and Aids

Conversion from octal to decimal is easy, if not fast. Here is a conversion of 02113 to decimal:

```
2 * 8^3 = 2 * 512      = 1024
1 * 8^2 = 1 * 64       =   64
1 * 8^1 = 1 * 8        =    8
3 * 8^0 = 3 * 1        =    3
                         -----
                         1099
```

The thing to remember is that the powers of 8 increase from 0 on the right, by 1 for each position to the left. You don't even need to remember what the numbers are themselves, just the numbering of the powers from the right.

Conversion of decimal to octal is almost as easy. Divide over and over by 8 until the quotient is zero. The successive remainders form the digits of the octal number, *from right to left*. Here is a conversion of 541 to octal:

```
541  / 8    = 67, remainder = 5
67   / 8    = 8,  remainder = 3
8    / 8    = 1,  remainder = 0
1    / 8    = 0,  remainder = 1
```

The result is 01035. Just for practice, what is the binary equivalent of 01035? Grouping the binary digits for each octal digit together, it is $001,000,011,101_2$.

While you do need to know the rules for octal arithmetic, you don't have to hand-convert every octal number you encounter. There are a number of ways to make the job easier:

- Use your creativity to avoid having to look at large quantities of octal numbers in the first place. Use the edit package (EDIT$/AEDIT$) to format output numbers in decimal. Use PMD's formatting options to display numbers in decimal, floating point format, ASCII, etc.
- Use the conversion tables in the back of your OS 1100 Programmer's Reference or Series 1100 Software Quick-Reference Guide.
- For numbers much larger than 4095 (octal 07777), you will need an octal/decimal calculator such as the Texas Instruments LCD Programmer or Hewlett-Packard 16C. Casio makes one, as well. In addition, a number of PC-based calculator programs (such as the one in the WordPerfect Library) can accept octal. Be warned, though, that few of these options can accept word sizes greater than 32 bits. (The H-P 16C, on the other hand, allows you to set the word size, and does ones complement and twos complement equally well.)

3.4 1100/2200 Operand and Instruction Formats

3.4.1 Data Formats

3.4.1.1 Fixed-Point Formats

Single-precision fixed-point integer is the simplest and most common data format you'll work with. It is 36 bits wide, and the leftmost bit is the sign (see Figure 3-2, following page). This is the same width as each of the X, A and R registers, 36 bits.

Double-precision fixed-point integer is the unit of data operated on by double-word instructions such as Double Load, Double Store, and Double Test Equal. It is 72 bits wide, and the leftmost 2 bits contain the sign (see Figure 3-3, following page).

3.4.1.2 Floating Point Formats

Single precision floating point is the unit used by most scientific programmers in Fortran. (See Figure 3-4, following page.) It can handle very large and very small numbers with equal ease. The leftmost bit is, once again, the sign. The next 8 bits form a field called the characteristic, the last 27 the mantissa. These last two fields are analogous to scientific notation. The characteristic represents an exponent, in this case a power of two. The mantissa is the fractional part of the number.

Since the main concern of this text is not scientific programming, you may want to refer to the 1100/90 Processor and Storage Reference (UP-9667 Rev. 1), section 3.18.7, for an excellent introduction to floating point arithmetic.

Double precision floating point is handled by the machine in much the same way as single-precision floating point. But since the characteristic and mantissa are both wider, double precision can accommodate much larger numbers with much more accuracy. The sign is the leftmost bit, the characteristic is 11 bits wide, and the mantissa 60 bits (see Figure 3-5, following page).

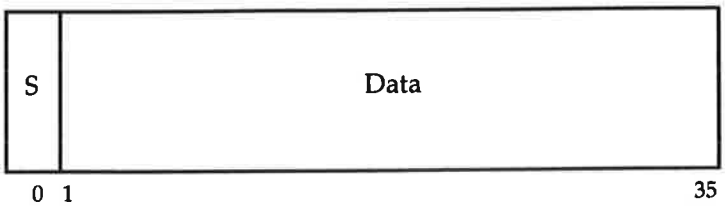

Figure 3-2. Single Precision Fixed-Point Data

Figure 3-3. Double Precision Fixed-Point Data

3.4.1.3 Characters

The second most common data type you will encounter in MASM programming, after fixed point binary, is character strings. MASM can automatically handle Fieldata or ASCII character data.

Fieldata (abbreviated sometimes as FD) is the older, 6-bit character set. Many Executive Request packets and other Exec-related information still use Fieldata, so make sure you are familiar with it. (See Figure 3-6 for the Fieldata format, following page.) Because there are only 6 bits in each Fieldata character, there are only 64 Fieldata characters. (Two to the 6th power is 64.) That means there are upper case letters only, not lower case, and a number of special characters are not available.

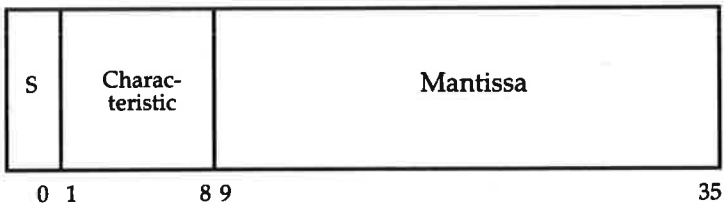

Figure 3-4. Single Precision Floating Point

The industry standard, and an emerging standard for 1100 software, is ASCII, an acronym for American Standard Code for Information Interchange. (See Figure 3-7 for the quarter-word ASCII format, following page.) One ASCII character consists of a

7-bit field (therefore 128 possible characters), normally extended with a zero on the left to 8 bits. The 1100 stores four ASCII characters per word, in Q1, Q2, Q3 and Q4, with the leftmost bit of each quarter word left as zero.

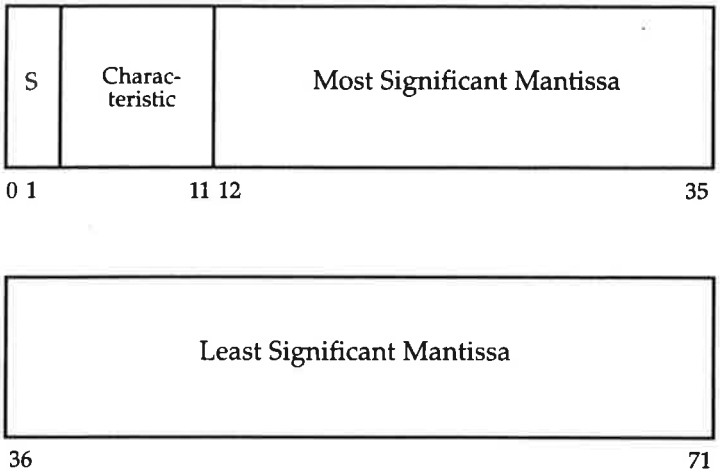

Figure 3-5. Double Precision Floating Point Format

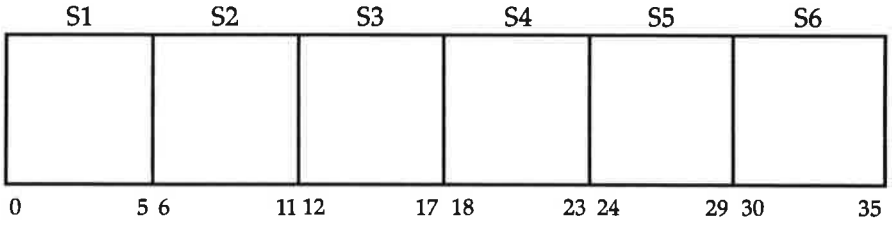

Figure 3-6. Fieldata Characters

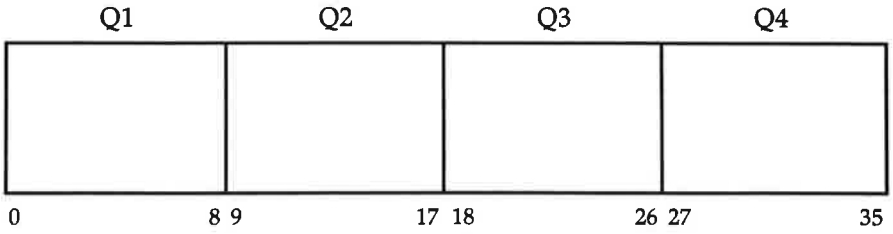

Figure 3-7. ASCII Characters

STYLE: If you can code most of your program to use ASCII, it will be easier to upgrade as the Exec is changed to use more and more ASCII.

Some applications, mostly using magnetic tape, have to work with 8-bit packed ASCII. It contains 9 characters in 2 words. The fifth character is split between the two words. (See Figure 3-8.)

There are conversion tables for both Fieldata and ASCII in Section 4 of your Operations and Programming Quick-Reference Guide. Look over them now, to familiarize yourself with their location and organization.

Chapter 21 tells more about how to access character data.

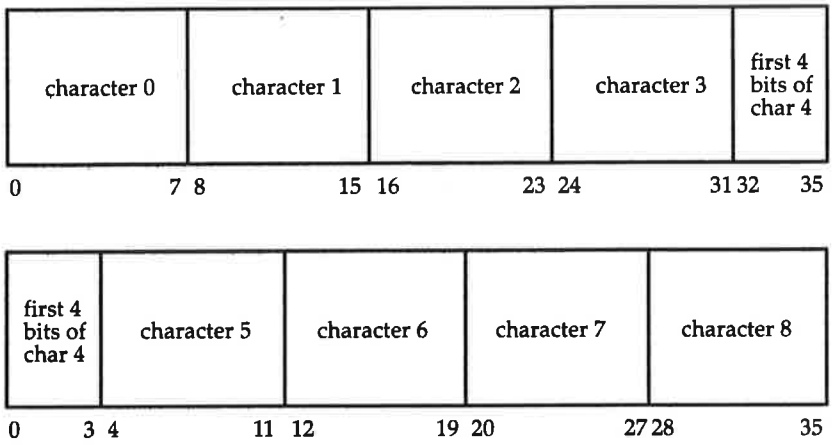
character 0
character 1
character 2
character 3
first 4 bits of char 4
0 7 8 15 16 23 24 31 32 35
first 4 bits of char 4
character 5
character 6
character 7
character 8
0 3 4 11 12 19 20 27 28 35

Figure 3-8. Eight-bit ASCII Characters

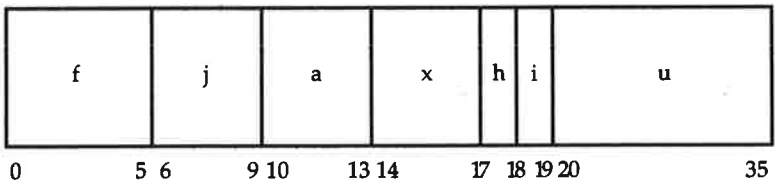

Figure 3-9. Series 1100 Instruction Format

3.4.2 Instruction Word Format

The instruction word has 7 fields: f, j, a, x, h, i and u (see Figure 3-9). The fields are, in order:

- *f* - Function or "op" (operation) code. This is the first field used by the CPU to determine the operation to be performed. It is 6 bits wide.
- *j* - Minor function code or partial word designator. For some f codes, this field is a continuation, further defining the operation to be performed. For other op codes, j specifies

which portion of an operand to operate on. (You can think of it as a "partial word index".) This field is 4 bits wide.

- *a* - Register designator. This field tells which register an instruction is to use. In the case of instructions which do not use an A register (such as Store ASCII Spaces), the a field acts as an extension to the f field or the combined f and j fields.

- *x* - Index register designator. This field tells the CPU which, if any, index register will be added to the u field when creating an effective operand address, U. If x is 0, the CPU does not perform indexing.

- *h* - Index register incrementation bit. This bit, when set, tells the CPU to add the contents of the x-specified index register's increment field to its modifier field. This feature is useful for stepping through repeated table entries.

- *i* - Indirect bit. This bit, when set, tells the CPU, once it has created an effective operand address, to take the rightmost 22 bits of that operand. From the new x, h, i and u fields, it generates a new effective operand address. If the i bit is set in the new operand, the CPU will repeat the process. The CPU can "cascade" (chain) indirect references in this way without any practical limit.

- *u* - Operand location. This 16-bit field supplies the location of the instruction's operand. It can be supplemented by the value of the specified x register.

Exercises

E3-1. Convert the following numbers from decimal to octal:

9
27
67
212

E3-2. Convert the following numbers from octal to decimal:

010
0100
033
01023
0777777777734 (Hint: it is negative.)

E3-3. In one of your reference manuals (figure out which; more than one has it), look up the addresses of the user registers and the Exec registers. Note the other registers in the address range below 0200.

E3-4. Find each of the word formats discussed in this chapter in one of your reference manuals. (Hint: there is more than one place to find them.)

E3-5. Look up in your Processor and Storage reference the use of the j, h and i fields. Even if you don't follow all the detail you find, note where you found the information.

Chapter 4: MASM Basics

4.1 The Fine Print

The following information about MASM is *not* intended as a reference to MASM, just an introduction to the basics. For more details, look at the MASM Reference.

4.2 Symbols

MASM symbols are strings of up to 12 characters, consisting of the characters A-Z, 0-9, $ (dollar sign) and _ (underscore). A symbol must begin with an alpha character (A-Z), except for symbols defined by the MASM processor. Distinctions between upper case and lower case characters are ignored in symbols, so xyz and XYZ are equivalent.

4.3 MASM Line Structure

Only one MASM statement may be placed on a single image, although a statement can be continued to multiple images. There are three major fields on a MASM statement:

1. Label,
2. Operation and
3. Operand.

Each field is optional. There is also a comment field, not scanned by MASM, but important nonetheless. The description of each field shows the field shaded.

4.3.1 The Label Field

```
<label>    <op>,<j>    <opnd>,<opnd>,...      . <comments>
```

There are three types of entities which can be used in the label field:

- Page control,
- A location counter specification and
- A label.

A page control symbol is a slash (/) in column 1; it causes a page eject. A location counter specification consists of a dollar sign, followed by a number or symbol in parentheses (like "$(1)"). The number or symbol must be between 0 and 63, inclusive.

The label itself is a MASM symbol which is normally defined with the current value of the current location counter. (The exceptions will be noted later.) The label may be followed immediately by an asterisk (*) to externalize the label, that is, define it outside of the program.

You can combine all three, page control, location counter specification and a label in the label field. But putting them all together makes the code a bit harder to read.

4.3.2 The Operation Field

```
<label>          <opnd>,<opnd>,...        . <comments>
```

The next non-blank character begins the operation field. It tells MASM what kind of instruction to generate. Or it may be a procedure (proc) reference or a MASM directive, which might or might not result in code generation.

For example, an instruction might be an LA, for "load A register". A proc reference might be AXR$, which defines register names such as A0, X5, etc.

A subfield of the operation field (<j>) can contain a partial word modifier, such as S5 for the fifth 6-bit division of a word. Chapter 7 discusses partial word modifiers.

4.3.3 The Operand Field

```
<label>     <op>,<j>                          . <comments>
```

This field supplies the remainder of the parameters used to generate an instruction word. If the operation field is a proc reference, the operand field normally supplies the proc's parameters.

For example, the operand field of an instruction might be "A5,TAG1,X10" (without the quotes). What these fields mean will be explained later.

4.3.4 The Comment Area, Column Conventions, and Continuation

```
<label>     <op>,<j>  <opnd>,<opnd>,...      . <comments>
```

The next non-blank character after the end of the operand field begins the comment field. However, since some procs may allow a variable number of specification fields, it is a good idea to separate instruction fields from comments using a period space terminator (". ").

STYLE: The recommended tab settings for assembly language are 11 for the operation field and 21 for the operand field. The period starting the comment field should be in column 39, and the first non-blank character of the comment should be in column 41. While it *is* allowed by MASM to separate fields with only one space, consistent column alignment is much easier to read.

If your statement is too long to fit on a single line, continue it to the next line with a semicolon (";"). If the continuation is in the middle of a field, put the semicolon right after the comma separating subfields. The next non-blank character continues the field:

```
PROC1       SPEC1,SPEC2,;
            SPEC3   .
```

STYLE: CTS users can put a TAB ^ 11,21,39 in their USER$ element for these tab settings.

STYLE: If your proc can handle a large number of subfields or fields, feel free to put them on consecutive lines. MASM is remarkably accommodating. For example:

```
<label>     TRANSLATETAB     '0',060  ;
                             '1',061  ;
                             '2',062  ;
                             etc.
```

4.4 Labels and Location Counters

MASM builds what is called a "dictionary" as it interprets your code. It consists of a list of labels and their attributes. The attributes are descriptors, such as:

- The value of the label.
- The location counter associated with the label (used for relocation purposes).
- A flag indicating if the label defines a directive (such as a $FORM, explained later) as opposed to ordinary data.

Ordinary MASM labels are simpler than variable names used in Fortran or Cobol. Define a label by putting it in the label field of a MASM statement. The label will be placed in MASM's dictionary for the duration of the assembly and is given the current value of the current location counter. The value of the label is the integer address of a location, not the contents of that location. This is one major difference between MASM and a high-level language.

There are three major exceptions to the "current value of the current location counter" idea.

- First, you can associate a value with a label using the $EQU directive:

```
NBRRUNS     $EQU          12          . SET TABLE SIZE.
```

An explicit $EQU like this is one of the few ways to redefine the value of a label, but only if the label was first defined with a $EQU. (MASM will generate error flags if a label is assigned two different values implicitly.)

- Second, the label on a $DO or $REPEAT directive is successively assigned the value of the loop counter on each iteration (more about this in Chapter 18):

```
I           $DO          15 , + I-1 .
```

- Third, a label on a procedure reference is not automatically assigned the current value of the current location counter. To ensure this action, use a "waiting label", described in Chapter 16.

A location counter is a type of counter MASM uses to control the entry of items into memory locations at assembly time. As MASM interprets your code, it adds instructions and data words to tables. Only one of those tables is active at a time; that is, MASM adds instruction or data words to only one at a time. The location counter specification tells MASM in which table to accumulate instructions or data. Since there is a one-to-one correspondence between counter and table, programmers often blur the distinction between them. We refer to a contiguous set of memory locations under location counter control as a "location counter". With practice, you will be able to tell which meaning is intended from the context:

- At assembly time, the term usually means the counter itself.
- At collection and execution time, the term usually means the contiguous set of data.

Note that you cannot increment or decrement a location counter directly. You affect a location counter indirectly by generating instruction and data words, or by reserving locations. How you do those things is the subject of the rest of this book.

MASM allows us to use up to 64 location counters (numbered from 0 to 63), but in practice most programmers need only a small fraction of that.

By a tradition dating back to 1108 hardware design, we use even location counters (0, 2, 4, etc.) to contain data. The collector by default puts them in the D (data) banks of an absolute element. We use odd location counters (1, 3, 5, etc.) for instructions. By default, the collector places these location counters in the I (instruction) banks of an absolute. The chapter on the collector will explain more about how this came about.

STYLE: Even though you may not need to split your program in multiple I and D banks, you can make looking at a post mortem dump easier by splitting data areas into several location counters. Place small work areas under $(0) (location counter 0) and large buffers into $(2), $(4), etc.

4.5 Register and Partial Word Definitions

Nearly all MASM programmers use the same register names, X1 through X11, A0 through A15, and R1 through R15. But MASM itself does not provide the definitions for those names, even though it does recognize 1100 instruction mnemonics by default. There are two ways to define the register names; choose one of these:

- The first, and oldest, is a procedure called AXR$. Place the proc name in the operation field.

```
        AXR$
```

- The other method is more efficient, and is tailored to use with MASM. MASM can create a special format omnibus element which contains a procedure definition, through "definition mode assembly". Someone has already done that step for you. To use the proc, put this line in your code:

```
        $INCLUDE    'MAXR$/'
```

The format of MAXR$ is less generalized, but fits into MASM's data structures directly, without interpretation, using less overhead.

Both AXR$ and MAXR$ are documented in Chapter 2 of the SYSLIB reference, UP-8728 (latest revision).

Both these procedures also define a set of tags you can use to reference partial words. Partial words are analogous to byte and half-word instructions on a byte-oriented machine. The 1100 data word, 36 bits long, can be divided four ways:

1. Halves 18 bits long, H1 and H2, and XH1 and XH2,
2. Thirds 12 bits long, T1 through T3,
3. Quarters 9 bits long, Q1 through Q4 and
4. Sixths 6 bits long, S1 through S6.

These divisions invariably start and end on word boundaries. They divide the 36 bits into equal parts. It is clumsy to use both third- and quarter-words in the same context, for reasons revealed later.

To reference a partial word, place the partial word name after the instruction mnemonic on an instruction line:

```
L,H2        A5,TRACKS  .
```

The partial word names, like the register names, become simple numbers inserted into instruction fields. Partial word transfers will be described in Chapter 7.

4.6 Data Generation

If you omit the operation field of a MASM source line, but not the operand field, MASM will implicitly call the $GEN directive to create a data word. The data is placed in the area associated with the current location counter. It will also increment the location counter. (The word "increment", unless otherwise noted, will be used to mean "add 1 to". "Decrement", likewise, will mean "subtract 1 from".) This line will create a word consisting of the value 5 (octal 000000000005):

```
5  .
```

But it is more conventional to put a sign in front of a constant, even though it is most often a plus sign. This line generates the same value, 000000000005, as the previous example:

```
+          5  .
```

A negative sign generates a negative value:

```
-          5  .
```

This line generates a value of 0777777777772. (The leading zero is not part of the number.)

If the operation field consists of alphabetic characters (like ABC or RES or UNLIST), MASM will interpret it as a instruction or directive, whichever is defined.

MASM can generate a word divided into almost any number of *equal* parts with + or - in the operation field. Two numbers together generates 2 half words:

```
        +       5,2     .
```

This example places 5 in the left half of a word (H1), and 2 in the right half (H2). Similarly, three, four or six values divides a word into thirds, quarters or sixths:

```
        +       7,-22,84        . THREE PARTS
        +       2,7,4,18        . FOUR PARTS
        +       0,1,-0,2,5,8    . SIX PARTS
```

MASM will reject a number of subfields which does not evenly divide a data word, say 5. (36 is not evenly divisible by 5.) The good news is that an even division of 36 would include 12 three-bit fields.

4.7 $FORM Directive

In other cases, you may need to divide a word into unequal fields, say, 12, 6 and 18 bits. Or 3, 15 and 18 bits. MASM provides a directive to associate a name with a special word division. Supply the $FORM directive with the label and the bit sizes of the fields:

```
PF      $FORM   12,6,18     .
ODD     $FORM   3,15,18     .
```

After you've defined the form, it becomes a legal operation. Put the form name in the operation field, and put the values to go into the bit fields in the operand field:

```
TAG     PF      10,27,BUFFER    .
```

Everything discussed so far has fit into a single 36-bit word, but the machine can work with double words, too. To generate a double word data item, use a postfix D:

```
DW5     +       5D      .
```

Or, if you forget the D, and the quantity won't use more than 36 bits initially, generate two single words, putting zero in the first:

```
DW5     +       0       .
        +       5       .
```

You can treat these two words as a single 72-bit field or two separate words, as necessary. The computer won't know the difference.

STYLE: If you put the label for a data table on a separate image from the first image of the data, it will make it easier to add data lines in between later on. This is especially true if you must make the additions via Source Input Routines (SIR) correction images of the form -n or -m,n. In fact, this is excellent advice for instruction images, as well. What follows is an example:

```
TABLE1      .
            +           6,5         . TABLE ENTRY
            +           15,0        . TABLE ENTRY
TLEN1       $EQU        $-TABLE1    . LENGTH OF TABLE
```

This technique makes it easy to add, delete or move table entries easily. Separating the label from the operation field with an editor can be messy if both are on the same line.

The vast majority of the data you will use in assembly language is integer, fixed-point binary, the only kind of data dealt with so far. The machine can also process floating point data, in both single and double precision. MASM recognizes floating point numbers by the decimal point:

```
FP1         +           1.0         .
```

The decimal point may be the first character, last character or embedded. Take care that a number starting with a zero and containing a decimal is not interpreted as a floating point octal number. Check your MASM reference for the details.

Once again, you can use the postfix D to tell MASM to generate double precision (two words). You will probably need to deal with floating point data only rarely.

4.8 Other Basic Directives

All of the MASM directives and functions used so far have two forms: redefinable, and non-redefinable. The redefinable forms lack the leading $ character. Their default definitions refer to the non-redefinable versions, which begin with a dollar sign ($). For example, REPEAT is initially defined as $REPEAT. If necessary, you could switch the ordinary meanings of REPEAT and DO:

```
REPEAT      $EQU        $DO         .
DO          $EQU        $REPEAT     .
```

although it is not recommended. More practically speaking, you could create your own Fieldata character set directive:

```
FD          $EQU        $FDATA      .
            FD                      . SWITCH TO FIELDATA
```

which brings us to the first MASM directive, $EQU.

A $EQU explicitly associates a label with a value. That value may be fixed point binary, string, floating point or a directive such as a $FORM. The latter is what makes this a META assembler. You can treat a proc name or instruction mnemonic as an ordinary symbol. MASM will create an error flag if you try to use a symbol associated with control information in the operand field:

```
CELL02      +           $EQU  .
```

MASM allows you to redefine the values of explicitly defined labels, that is, labels defined with $EQU. The other way of defining a label, implicitly, such as placing a label in the label field of an

instruction, is irrevocable. (In other words, don't attempt to redefine a label which has been defined implicitly. You will get a D assembly error flag.) Here is an example of explicit redefinition:

```
TMPCON      $EQU        7           ; ORIGINAL VALUE
WORK10      +           TMPCON      ; USE ORIGINAL VALUE
TMPCON      $EQU        100         ; NEW EXPLICIT VALUE
```

The first $EQU gives a temporary constant, TMPCON, its first value, 7. The second $EQU gives it another value, 100. MASM accepts this explicit redefinition. Here is an implicit redefinition:

```
WORK88      +           5           .
...
WORK88      -           CONST9      .
```

MASM will reject this code with an assembly error flag.

The next basic directive is $RES, short for reserve. $RES tells MASM to increase the current location counter. Typically, a $RES directive will be on the same line as or immediately following a label. Since the label will be associated with the current value of the location counter, the label will point to the reserved area:

```
TABSIZ      $EQU        5           ; NUMBER OF ENTRIES
TABENT      $EQU        20          ; ENTRY SIZE
TABLE2      .
            $RES        TABSIZ*TABENT .
```

In this case, TABLE2 is the address of the table, which will be TABSIZ words per entry times TABENT entries long.

STYLE: Parameterize everything you can. If you code a $RES statement for 100 words, and have to change something a year later, you could be in trouble. How big is each entry? How many entries did you intend to leave room for? How many ways are there to factor 100? Coding $RESes with $EQUed labels as shown above makes maintenance much easier, for both you and somebody down the line. Also, the code which uses the table can use the same $EQUed labels to scan the table, ensuring that it never oversteps the bounds of the table.

Another directive you need is $END. It has two functions:

- First, it terminates the current subassembly. If the current subassembly is a procedure definition, the $END is the end of the procedure. If not, the $END is the end of the source input.

- Second, the $END directive is the only place you can use to tell MASM to generate a start address, the place to begin execution. (This convention goes back to the early days of computers; the last statement had to be a jump or branch to the beginning.) The start address in the operand field is optional. If you omit it, MASM will not put a start address entry in the output relocatable element.

A $END directive normally has no label. Here is an example:

```
        $END    START   .
```

After loading the program which contains this element, the Exec will pass control to the label START.

Not every element needs a start address. Only one element in a collection may have a start address, or you may have to resolve the ambiguity with a collector directive. (See Chapter 22 for more about the collector.)

4.9 Character Strings

4.9.1 Character Set Control

Control the way MASM processes your character strings by using the $ASCII and $FDATA directives. If the $ASCII directive is in effect, it will convert your character strings into ASCII:

```
        $ASCII          .
```

If the $FDATA directive is in effect (and that is the default), MASM will create Fieldata character strings:

```
        $FDATA          .
```

4.9.2 EQUating Character Strings

You can associate a label with a character string using the $EQU directive:

```
NAME    $EQU    'MAX PAGES' .
```

Note that this does not put the character string into data words. It is effective only at assembly time. It does allow you to manipulate characters, concatenating them with the colon operator:

```
PROG    $EQU    'QASG'      .
MSG1    $EQU    PROG:' ERR TERM: CALL SUPPORT STAFF'
MSG2    $EQU    PROG:' NORMAL TERMINATION' .
```

Better still, the value of PROG can be made dependent on another variable, changing the text strings according to configuration variables.

To put a character string into data words, put it in the operand field:

```
TEXT1   'PLEASE MOUNT TAPE NOW' .
```

The MASM manual calls this an unsigned character string, and it is the most common. MASM will create enough data words to hold the string, with the first character left-justified in the first word. MASM will pad the last word with spaces as necessary. This format is called LJSF (left-justified, space filled). Create a single quote in a string by using pairs of single quotes:

```
NEGACK      'YOU CAN''T DO THAT!' .
```

4.9.3 Signed Character Strings

MASM can also create a signed character string of no more than 2 words. It is right-justified and zero filled. You will use it most often performing fixed-point comparisons on character data. For example:

```
            TE,U        A3,'Y'      . Operator answered yes?
```

Note that this string constant is 18 bits long.

STYLE: If you expect to test single characters in your instructions, make sure that the last character set directive in effect in your D bank code is the one you want in effect for the instructions. If you are testing for a Fieldata value, make sure that a $FDATA directive is in effect; if you are testing for ASCII data, make sure a $ASCII directive is in effect.

The adventurous programmer may want (or need) to create a whole new character set, or create a variation of ASCII or Fieldata. It may be necessary to replace only a single character. Or, creating a complete character set might be necessary to build a tape for communicating with a foreign, non-ASCII system. Use the $CHAR directive to do either job. The details of this unusual directive are in the MASM reference, section 4.2.3.2.3.

4.9.4 String Functions

Where a MASM directive is a character string placed in the operation field, a MASM function is more similar to a Fortran function. The name of the function, together with a pair of parentheses enclosing an expression, is used within another expression. String conversion is one of the few places the average MASM programmer will need a MASM function. If a $FDATA directive is in effect, you can create an ASCII string:

```
LABEL       $CAS('CAN''T READ TAPE - WRONG FORMAT') .
```

The conversion function is $CAS (Convert to ASCII String). There is also a $CFS (Convert to Fieldata String).

The function $CD (Convert to Decimal) is useful, too. Say you need to convert an assembly-time value for display in a message:

```
REELSZ      $EQU        25          . DECIMAL PARAMETER
WARN01      $CD(REELSZ)             . STRING EQUIVALENT
```

The $CD function converts REELSZ from a binary assembly time variable into a character string of a decimal number (e.g., '25').

You may someday need to write a proc which scans a character string one character at a time. The function $SL (String Length) returns the number of characters in the string. The function $SS (Sub

String) extracts a portion of its string parameter. Here is an example using both (don't worry if it's a bit advanced for now):

```
STRING1     $EQU        'XYZABC'    .
SSN         $REPEAT     $SL(STRING1)  .
            +           $SS(STRING1,SSN,1)  .
            $ENDR                   .
```

This example steps through the characters of STRING1, and places each in memory, one character right-justified per word. Of the parameters supplied to $SS, the first is the string to extract from, the second is the starting character number and the last is the number of characters to extract. (Once again, check the MASM manual for the fine print.)

Exercises

E4-1. For each of these strings, which can you use as MASM symbols?

AN$GEL
$PUSH
aSHORt
_TAG27
TAG_LAST
7PHASE

If the third label were acceptable to MASM, would it be equivalent to the label ASHORT?

E4-2. Write the code to reserve space for a table of 100 widgets (hint: WDGCNT will be 100), where each widget is 5 words long (WDGLNG), but the length is subject to change.

E4-3. Code one character equate for each of your names (first, middle and last), then a line to store them as ASCII under location counter 4. Don't forget to insert spaces between the names.

E4-4. Write a string function to extract the third, fifth and eighth characters of your name, and store them in memory.

E4-5. For "extra credit", repeat E4-4, but store the extracted string as Fieldata. (Hint: you can nest string function references.)

Chapter 5: Assembling and Running your Program

5.1 Basic ECL Statements

Before you can get the machine to run your inspired creation, you've got to go through the operating system. Executive Control Language statements tell OS 1100 which files to assign, which programs to load, and other things which affect your programs.

The first statement the system sees is your RUN statement. It identifies you and your run to the system and sets limits on the resources you can use. For example, the account number on the RUN statement identifies who will be charged for the resources used by your run. Here is a sample RUN statement:

```
@RUN  CMHMDM,5048SYS/CMHSYS,SYSPROG,5,100  .
```

This run statement has the following fields:

- CMHMDM is the runid. This field is from 1 to 6 characters long, and the characters must be letters or numbers. (For example, $ and - are not allowed.) If the system finds illegal characters in the runid, it will ignore the rest of the runstream with little or no comment.

- 5048SYS is the account "number". This field is from 1 to 12 characters long, must contain only letters, numbers, hyphen ("-") or period (".") and is assigned by your system administrator. Some sites use multiple account numbers per group according to the task or project.

- CMHSYS is the userid. The userid uniquely identifies one specific user (you). It is from 1 to 12 characters, it must contain only letters, numbers, hyphen ("-") or period (".") and it is assigned by your system administrator. In most OS 1100 installations, each userid is allowed to use only certain account numbers.

- SYSPROG is the project field. It classifies runs and files within the system according to tasks or groups of jobs to be done. The project is the default qualifier for files your run assigns or catalogs. The project is from 1 to 12 characters, including the letters, the numbers and the characters $ (dollar sign) and - (hyphen).

- The 5 is the estimated SUP time in minutes. If a batch run exceeds this quantity, at many installations, OS 1100 will terminate your run in error. This time is measured in use of system resources, not in terms of wall-clock time. Your MASM programs should have a fairly small time limit, to catch the occasional infinite loop.

- The 100 is the estimated page count. Once again, on many systems, the operating system will terminate a batch run in error when it exceeds this page count.

If your run duplicates the runid of another job (or your own if it has not printed yet, for example), the system will concoct a "generated" runid from your runid. For runids less than 6 characters, OS 1100 will attach the letter A, like ADAMA from ADAM. The next such duplicate will be ADAMB, and so on. For runids of 6 characters, the system will step the last letter of the runid to the next letter

of the alphabet up to Z, then 0 through 9. Under these rules, CMHMDM would become CMHMDN, CMHMDO, and so on, through CMHMDZ, then CMHMD0, and so on. The next statement you will need to run your MASM program is @MASM. The general format is:

```
@MASM,<opts>   <sym_in>,<rel_out>,<sym_out>
```

Under OS 1100's dropout rules, if your source code follows the @MASM in the runstream, you can omit the options and all the specification (spec) fields. The statement is then simply "@MASM" beginning in column 1. MASM will place the relocatable output into TPF$.NAME$ (more dropout rules), but without assuming any listing options. These are the fields in the pointed brackets above:

- <opts> are the assembly options. I means the source code is coming from the runstream and MASM must create a new source element; S means a standard source listing; L means that MASM must print the most detailed listing available. These are the most important options. The S option is used most commonly. The L option listing may include one or more extra listing lines for each generated data word, showing such details as which bits are modified by which location counter or external reference, and so forth. Omit the I option if you have created the source code with an editor and you want MASM to read the source from there.

- <sym_in> is the symbolic input spec. If you have specified the I option, this is where MASMwill place the source after reading it from the runstream. (If you unintentionally give MASM the I option, this is the name of the element you will have to re-enter after MASM wipes it out!)

- <rel_out> is the relocatable element specification. This is where MASM will place the results of the assembly. The collector will need this element to create an absolute element you can execute. If you omit this spec, MASM will use either the same file and element name as <sym_in>, or if <sym_in> is omitted, TPF$.NAME$.

- <sym_out> is the symbolic output specification. You will need this only in unusual cases; do not use it unless you know what you are doing. (After MASM applies source corrections to <sym_in>, it puts the updated symbolics in <sym_out> if <sym_out> is specified.)

Next, you will need to tell the system to take the relocatable element produced by MASM and create an absolute element which the system can load and execute. That is the job of the collector, or the MAP processor (they are one and the same). Here is the general form of the MAP processor call:

```
@MAP,<opts>  <sym_in>,<abs_out>,<sym_out>
```

These are the fields on the MAP processor call:

- <opts> is the option field. The I option tells MAP to create a symbolic element from the map source statements which follow. Again, like MASM, the S option requests a standard listing, L the most detailed, and N no listing at all. Other options control actions taken on error conditions or the way the output absolute element is created. Until you are experienced, you may want to stick with listing control options.

- <sym_in> is the file and element where MAP places the symbolic directives (if I option) or obtains symbolic directives (no I option). MAP will always look for symbolic

directives, or if there are none, it will collect the relocatables in TPF$ and create an absolute element named TPF$.NAME$.

- <abs_out> tells MAP where to place the absolute element it has created. If omitted, MAP will place it in TPF$.NAME$.

- <sym_out> is used to specify the destination of updated source code. You probably don't need to use this field.

Some programmers make the mistake of using the same symbolic element name on a @MAP,I image as on the @MASM image. @MAP,I will cheerfully create a new map symbolic element with the same name as your MASM source, wiping out your MASM source. Don't make this mistake!

Next, you need to tell the system to execute your new absolute element (that is, program). Use the @XQT statement:

```
@XQT,<opts>   <abs>
```

The fields are:

- <opts> are option letters (A-Z) available to the program (by means of OPT$). The system does not otherwise act on this field.

- <abs> is the name of the absolute element to load and execute, in OS 1100 qual*file.elt/vers notation. The usual dropout rules apply. If you do not specify a file or element name, the system will load the absolute element most recently added to TPF$ (usually NAME$).

5.2 Example ECL Streams

Example 1:

```
@RUN  <fill this in yourself>
@MASM
 <source program here>
@XQT
 <data here>
```

This is the minimum possible runstream. It uses the system dropout rules to the maximum:

- MASM will read the source from the runstream and create a relocatable element named NAME$ in TPF$.

- The system will see the @XQT with no absolute element named and no absolute in TPF$, and implicitly call the collector.

- The collector will collect whatever relocatables exist in TPF$, including NAME$.

- Then the system will execute the absolute element most recently added to TPF$, once again named NAME$.

This operation will leave two elements in TPF$, each named NAME$: a relocatable and an absolute. (Note that this sequence only works once. Another minimal @MASM will call the assembler, but the second @XQT would find the original absolute and execute it.)

Example 2:

```
@RUN  <fill this in yourself>
@MASM
 <source program here>
@MAP,L
@XQT
 <data here>
```

This example would behave very similarly to Example 1, except that it calls the collector explicitly, and asks the collector for a detailed listing. The element names will be the same (all NAME$).

Example 3:

```
@RUN  <fill this in yourself>
@MASM,IS   MYFILE.MYPROG
 <source program here>
@MAP,S     ,MYFILE.MYPROG
      IN   MYFILE.MYPROG
      NOT  TPF$.
      END
@XQT       MYFILE.MYPROG
 <data here>
```

This is a less minimalist ECL stream. MASM puts the source and relocatable into MYFILE.MYPROG, and generates a standard listing. The collector will collect the relocatable element MYFILE.MYPROG and create an absolute with the same name. The comma before MYFILE.MYPROG on the MAP call tells the collector not to bother creating a map symbolic element (the <sym_in> specification is empty). The map directive NOT TPF$ overrides MAP's default attempt to collect everything in TPF$. (This statement is a good idea in *any* map which does not really need to collect everything in TPF$.)

Example 4:

```
@DELETE,C  PRT1.
@ASG,UP    PRT1.,F///200
@BRKPT     PRINT$/PRT1
@HDG ** ASSEMBLING MYPROG **.M,66,2,2,6
@MASM,S    MYFILE.MYPROG,MYPROG
@MAP,IS    ,MYFILE.MYPROG
      IN   TPF$.MYPROG
      NOT  TPF$.
 END
@XQT .....MYFILE.MYPROG
 <data here>
@BRKPT     PRINT$
@FREEPRT1.
@SYM       PRT1.,,PR
```

This example assumes that you have already created the symbolic element in MYFILE, using CTS or the IPF-1100 editor. The ECL shown tells MASM to obtain the source code from MYFILE.MYPROG and put the relocatable into TPF$.MYPROG. The collector directives include the seemingly contradictory IN TPF$.MYPROG and NOT TPF$. The NOT TPF$ has exactly the same effect as the previous example, overriding the default collection from TPF$. The IN TPF$.MYPROG tells the collector to collect that relocatable from TPF$. No, MAP does not get confused or upset.

This type of ECL stream is the model for nearly everything the author does. This way, you keep only the source and absolute for each program on file; if you have the source, you can always re-create the relocatable. With modestly-sized programs, you can do this kind of work for several hours before using the default maximum space in TPF$.

There are also several new lines slipped in. The first four lines delete and recatalog a print file to breakpoint into, then breakpoint into it. The last few lines end the breakpoint and send the file to the printer.

Or, you can use an editor to scan the print file in demand mode. Look for lines beginning with the string "END". Most processors in the system produce summary lines of the form END <processor>. They tell how successful the operation was.

Many sites have a processor which automatically creates a print breakpoint file and does the @BRKPT into it. A matching processor ends the breakpoint mode and can optionally call the editor or send the print file to a printer. Check with your system library maintenance person.

The .M,66,2,2,6 on the @HDG sets 66 lines per page, top and bottom margins of 2 lines, and 6 lines per inch. These margins get more printable lines per page without undue crowding.

5.3 Listing Format

(See page *xi*, back of the book, for a small sample assembly listing. Refer to it as you read this description of the MASM listing format.) The main part of MASM's output is divided into four regions:

- Assembly error flags, if present, begin in column 1.
- Generated code, listed in octal. The breakdown of this area is listed below.
- Line numbers. If you are using SIR correction lines, you can give MASM the V option to request two sets of line numbers. One is the input or "base" line numbers; the other is the updated line numbers.
- The source lines.

The generated code, the left 25% of MASM's print output, is divided differently according to the word format, data or instruction. This division helps you to make sure that MASM interpreted the source line the way you intended. Here are some of the formats MASM uses in the generated code part of the listing:

- $EQUs are always listed as 12-digit octal numbers